DEBUGGING TECHNIQUES

I.P. Sharp Associates

I.P. Sharp Associates

ADVANCED *APL* TOPICS

DEBUGGING TECHNIQUES

I.P. SHARP ASSOCIATES

SHARP APL EDUCATION

BRIAN TRAQUAIR

TABLE OF CONTENTS

Contents	Page No.
DEBUGGING TECHNIQUES	1
INTRODUCTION	2
THE SOFTWARE "BUG"	3
. TYPES OF BUGS	4
. CAUSES OF BUGS	6
METHODOLOGY OF DEBUGGING	7
GENERAL DEBUGGING METHODS	9
SHARP APL TECHNIQUES	10
DETECTION	
. GENERAL METHODS	11
. SPECIFICS	12
LOCATION	
. GENERAL METHODS	13
. SPECIFICS	14
EXPLANATION	
. GENERAL METHODS	16
. SPECIFICS	17
CORRECTION	
. GENERAL METHODS	19
. SPECIFICS	20
GOOD PROGRAMMING PRACTICES	22
. SYSTEM DESIGN	23
. CODING	24
. TESTING	26
. DOCUMENTATION	27
CONCLUSION	28

ADVANCED *APL* TOPICS

DEBUGGING TECHNIQUES

"In a few minutes a computer can make a mistake so great that it would take many men many months to equal it."

Merle . Meacham

"Murphy's Law: If anything can go wrong, it will. "

"Program Complexity grows until it exceeds the capability of the programmer who must maintain it."

Osborn's Law: Variables won't; constants aren't."

"Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited."

"Weinberg's Second Law: If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

"Lubarsky's Law of Cybernetic Entomology: There's always one more bug."

INTRODUCTION

Computer programmers are without question the most optimistic and yet cynical group of professionals you are ever likely to meet. Each one wholeheartedly believes that the most recent error uncovered in his or her system is the last one and that the applied correction to the program will fix the problem once and for all. Each run is the last one -- there will be no more problems. Comprehensive testing is ridiculous - why shouldn't it work? Documentation is superfluous - the code is simple and obvious. The optimism goes on and on.

At the same time, this same group of people retain the most cynical attitude imaginable toward perfection in the computer industry. You have merely to read the previous quotes to see this clearly. Nearly everyone associated with this profession are fervent believers in and proclaimers of Murphy's Law (see quotes). No one apparently believes in the perfect system.

The subject of all this frustration, the focal point of much effort of professionals in the field and indeed our topic today is that arch nemesis of all computer related systems - the software "BUG"!

It is the objective of this seminar to detail the methodology of "DEBUGGING". This includes:

- defining the problem
- outlining methods of solution
- investigating available tools
- proposing better coding techniques

The methods are generally applicable to any programming language but this seminar is directed specifically toward the *APL* environment of I.P. Sharp Associates.

THE SOFTWARE "BUG"

A DEFINITION

Dictionary: a minor defect, as in a machine

Common Meaning: any uncorrected error in the coding of a computer system which causes incomplete and/or incorrect results

- often a "bug" is differentiated from a simple error by the greater difficulty encountered in detecting, locating and correcting it.

- also, "bugs" (minor though they may be) continue to crop up in a production system long after the testing stage has been completed

TYPES OF BUGS

1) errors which prevent a computer system from initiating its job

   eg. a missing file

   - these are usually more obvious than most, although the reason may be varied, and are relatively easy to correct and restart

2) errors which stop the execution of the system

   eg. branching to a non-existent label in a program

   - these are somewhat more difficult to find (except in APL) and may require complicated restarting

3) errors which prevent a system from terminating

   eg. an infinite loop

   - these are always eventually found but tend to be more expensive in terms of computer time

4) errors which allow the system to run but give incomplete and/or incorrect results

   eg. an incorrect calculation set

   - these are difficult to detect and often time-consuming to debug

   - they are also the worst kind of error when it goes undetected for a long period of time

5) errors which may or may not terminate the system but are data dependent

   eg. division by zero

   - these are the most difficult to debug because error may come and go from run to run

6) errors (rare!) in the data centre software or system utilities

   eg. an incorrect sort utility

   - these (although rare) do occur and cannot be neglected

7) errors (even rarer!!!) in the hardware used by the data-centre for computing

eg. a disk error

- these are the last place to look, depending on your equipment

CAUSES OF BUGS

1) carelessness in all phases of programming

   - this can be seen in the common exclamation, "Oh, of course! How stupid!"

2) typographical mistakes

   - related to an over reliance on the computer to catch spelling mistakes and inaccurate transcription from notes

   - most can be avoided by careful reading of the program

3) poor or non-existent testing procedures

   - once again, programmer optimism triumphs over common sense

4) lack of communication between development team members or between the designer and the end user

   - this type of problem increases exponentially with the number of people involved in the project (and hence usually with the complexity of the task)

5) incorrect assumptions concerning language constructs, system functions and utilities

   - due to reluctance to study documentation or perhaps, lack of documentation

METHODOLOGY OF DEBUGGING

DEFINITION: A methodology consists of the principles of orderly thought or procedures applied to a particular branch of learning.

Many programmers regard good debugging as more of an art rather than a science. This is incorrect. While intuition does play a part in the debugging process, the approach to the problem is well defined. The methodology consists of four steps:

1) DETECTION
2) LOCATION
3) EXPLANATION
4) CORRECTION

This procedure can and should be followed in attempting to solve any programming problem regardless of environment or complexity. The only variable is the duration of each stage according to person and problem. This approach provides a framework within which experience, concentration and intuition can play fruitful roles. The more elusive bugs may require several passes at each stage to finally correct the problem once and for all.

Briefly, a description of the steps:

Detection

- the discovery that an error has occured by looking either at the results or the system diagnostics

- hopefully this happens during testing, but more often than not it does not

Location

- determining the offending section of code within the system

- this may be one line, a subroutine or an entire program

Explanation

- deciding why the error has occured so as to ascertain its effect on the remainder of the system and to prevent it from cropping up again

- this is a very important part of gaining valuable experience during debugging

Correction

- formulating and implementing the necessary changes to fix the problem (and all similar ones)

- this may also involve restarting the program and other details such as documentation

GENERAL DEBUGGING METHODS

There are four stages inherent in the solution of any programming error. As we discuss the very specialized tools which enhance the debugging process in the *SHARP APL* environment, it is important to investigate the principles which apply to all programming languages and man-machine interfaces.

We shall do so by looking at each stage separately:

SHARP APL TECHNIQUES

The characteristics of the APL language and of the *SHARP APL* environment in particular are ideally suited for program development and debugging. Some of these advantages are:

- SIMPLE SYNTAX
- MODULAR STRUCTURE
- ARRAY ORIENTATION
- RICH SET OF PRIMITIVES
- CONVENIENT DEFAULTS
- INTERPRETIVE EXECUTION
- EXPLICIT ERROR MESSAGES
- RESTARTABILITY FEATURES

All of these items contribute to an easy to use programming language but the error handling capacity of the *SHARP APL* interpreter is of fundamental importance. It allows you to see exactly where execution of your program was interrupted, correct your mistake and continue the run. We shall discuss these features of *SHARP APL* , and others as we examine the tools and techniques useful in debugging.

In addition, the staff of I.P. Sharp Associates have developed, over the years, a number of workspaces of debugging and documentation aids. These packages have been placed in the public libraries for general use and we shall be pointing out where and how these should be applied.

S T A G E I

DETECTION

DETECTION

1) test all programs rigorously

   - few errors announce themselves - it is necessary to ferret them out by exposing the system to conditions and parameters which are likely to materialize

2) be on the lookout for variations in the run time or cost of a program without variance in volume of data

   - this is where data dependent bugs often show themselves

   - also the habitat of infinite loops and other costly mistakes

3) compare results of any modified program with its previous output

   - the chances of an error are higher than in a normal testing run

4) look for patterns or repetitions in the output

   - this can often be a sign of incorrect looping in a control structure

5) look for anomolies or exceptions in the output

   - may be due to incorrect data or indeed an error may have been uncovered

6) check the input data to ensure it is not the reason for strange results

   - the user created input may be outside the system's parameters of acceptance

   - a program can only detect errors it recognizes

7) obviously, a halt in execution accompanied by an error message is an indication of a programming oversight of some kind

   - if all errors announced themselves in this manner, the job of debugging would be much easier

8) ask others to test, read and use your system

   - others are not working under your assumption restrictions

   - this type of testing can be very revealing and also disconcerting when deficiences are pointed out

DETECTION

1) test each *APL* function separately and comprehensively to detect omissions in the code before it is part of a system

   - the interactive nature of *APL* makes this attractive and quick

   - there is no need to wait for batch turnaround for results or create files of data (usually) for testing purposes

2) most errors are caught immediately by the *APL* interpreter which makes detection straight forward

   - take caution to not rely on this facility completely for debugging

3) system imposed (and user specified) CPU limits and maximum file reservation quotas prevent runaway systems to a certain extent

   - it is the responsibility of the user to set reasonable constraints

S T A G E 2

LOCATION

LOCATION

1) the location of the error in the output may give a clue to the operation being performed , and hence the code to check

   - this can save a great deal of time and is sometimes dependent upon familiarity with the system

2) new or changed pieces of code are the first places to look upon encountering a problem

   - this can shorten the debugging cycle in many cases and is sometimes a matter of intuition as to the exact cause before investigating

3) if a production system, ask the user what he/she was doing when the problem occured

   - if at all possible, get a transcript of the terminal session or the input transactions

4) use the "SCIENTIFIC METHOD"

   - this means alter only one variable (data, program, parameters) and rerun the system to note diferences

   - in fact this is the fundamental way to track down bugs which have been detected but not found

   - it is crucial to alter only 1 thing at a time or the effort is useless

5) trace the system as it runs to gain insight

   - the method varies but is usually available in some form

6) print out intermediate results to check whether the output of a specific routine is correct

   - once each section is checked, it can be dismissed as the cause

   - sometimes this is done by stopping the system at suspected problem areas and inspecting the state of the program

7) compare and discuss your algorithm with others

   - by the time you explain your problem, you will have deduced the answer!

   - often a fresh opinion is the only alternative with a reasonable chance of success (due to frustration)

LOCATION

1) *APL* error diagnostic facility

- without question, this saves more debugging time than any other single characteristic of the language

- each error report gives:

  - type of error
  - name of function
  - line of code where error occured
  - a caret (" ^ ") pointing to the problem

eg.

```
      6   VALUE  ERROR
REPORT [3] DISPLAY←ITEMS BESIDE TOTAL
                                ^
```

2) batch error handling

- during *N-TASK* or *B-TASK* runs, an error (if not trapped) will halt execution and cause the workspace to be saved

- this workspace can then be loaded to discover the problem

3) *ΔTRACE* feature

- the *ΔTRACE* facility permits the programmer to follow the execution of any function desired

- simple syntax is to assign a variable, whose name consists of '*TΔ*' suffixed by the function name, a vector of line numbers to be traced

  eg. to trace lines 3 to 15 of function *DRIVER*

```
TΔDRIVER←2+⍳13
```

  to stop tracing ∘∘∘

```
TΔDRIVER←⍳0
```

4) *ΔSTOP* feature

- the *ΔSTOP* facility permits the programmer to halt the execution of any function desired

- simple syntax is to assign a variable, whose name consists of '*SΔ*' suffixed by the function name, a vector of line numbers before which it is to stop

eg. to stop before lines 6 and 8 of function *DRIVER*

```
      SΔDRIVER←6 8
```

to remove stops ∘∘∘

```
      SΔDRIVER←⍳0
```

NB. the trace and stop settings are valuable tools when debugging in attempts to track down an error in a function but should be used judiciously

- try to pinpoint the probable functions and trace them only (to avoid reams of output)

- when a stop occurs, display key variables to gain insight into the events taking place

5) *□OUT* setting to redirect output to a file or terminal as needed

- if reports are going to file, redirect them to the terminal for immediate examination

- if large volumes of error diagnostics are being displayed, put them to file to be submitted to *HSPRINT* and examined later

6) check *□TRAP* definitions and modify if necessary

- the trapping of events may hinder normal debugging methods so steps must be taken to combat this problem

  - incorporate a trace facility into the definitions to follow the flow of control

  - disable the definitions (if possible) while testing

NB. this topic is covered in more detail in "Event Trapping" (another Advanced APL Topic)

S T A G E 3

EXPLANATION

EXPLANATION

1) read the offending code to make sure it says what you originally wanted

   - typo's and transpositions are still the most common reasons for errors first time through

2) verify your assumptions as to the expected input and desired output of the routine

   - the passing of parameters or arguments can lead to errors when no meeting of minds exists

3) test the routine on its own outside of the system

   - only in this way can you experiment in a cost and time effective manner

4) consult documentation to discover any areas of the code in which you were incorrect

   - the last place people look, but often the most productive

5) if the reason for the error involves others, be sure to discuss it with them for their side of the story

   - misunderstandings are seldom one person's fault

   - don't guess at another person's intentions or program - ask them if possible

6) understand the whole problem, not just the apparent error

   - it is a common mistake to see a small error and fix it without condidering the larger bug which may be present

   - this is especially prevalent in an interactive programming shop

7) analyse the problem case by case in a stepwise approach

   - consider each class of input and output one at a time carefully

   - not only will this point out the error but it will give new direction for the modified code

8) ask someone to explain the problem to you

   - do not give up at once but do not let your pride hold up progress unnecessarily

   - always bring everything that will aid the search for an answer - listings, input data, results, etc.

EXPLANATION

1) Use ⎕ and ⍞ to display intermediate results within an incorrect line to determine why it is wrong

   - this is especially valuable when doing many operations on one line which are prone to errors

2) comment out suspect lines and replace them (or leave them out entirely) to note any effect

   - this is better than simply erasing them or replacing them in the function

   - allows you to keep track of the original error

3) execute the suspected code in immediate execution mode, step by step and check your results against your expectations

   - especially test the code you know is right, it is probably wrong

4) try alternate algorithms to see if you get the same result using a different approach

   - this is usually quick and easy to do in *APL*

5) consult documentation on the primitives and/or system functions being used to ensure you understand them

   - the place to look is in the "SHARP APL REFERENCE MANUAL" by Paul Berry of I.P. Sharp Associates

   - this book is an excellent reference book and can be used to great advantage

6) document your workspace(s) using either

7 WSDOC & *7 XREF*

or

777 WSTOFILE & *777 WSDOC*

   - these software aids allow you to specify a number of helpful options such as a cross-reference of variables and functions, in a listing of your workspace environment

   - this type of program listing is invaluable in checking systems for oversight and errors

NB: - documentation is on-line
    - *LIBRARY* 777 is a development library and therfore not warrentied although supported

7) further documentation facilities can be found in

777 WSGRAF & 777 FLOWCHARTS

- these workspaces provide a pictorial debugging tool by charting the flow of control within a large and complex system

- documentation is on-line

8) use *MAILBOX* to get extra help from I.P. Sharp personnel and others well versed in *APL*

- you must be enrolled to use *666 BOX*

- groups such as *HELP* and those with special interests can offer good solutions to well defined problems

S T A G E 4

CORRECTION

CORRECTION

1) test your correction to the routine in question independently of the system

   - prevents needless extra work due to noncomprehensive solutions to the problem

   - much cheaper and quicker than production runs

2) attempt to catch and correct all errors of the same type at the same time

   - very often erroneous code is replicated within a large program and its associated routines

3) anticipate the implications of any design or code changes you make before they happen

   - in any sample system, routines are often called by many programs, each of which may require slightly different specifications

   - assumptions incorrectly made for one routine may be followed elsewhere

4) instead of just recoding one solution, try to think of a new algorithm as an alternate

   - a second chance to code should not be passed up as an opportunity to improve upon oneself

5) perform a "WALK-THROUGH" of your proposed solution to ensure it meets all requirements

   - probably the best bug preventative technique of all!

   - a walk-through with pencil and paper forces you to evaluate your own solution reasonably objectively

6) compare the results of the first new production run with both the old results and your expectations

   - never assume anything will work without verifying it for yourself to your own satisfaction

7) whenever possible, restart the system after making corrections to minimize impact to the user

8) update program and user documentation to reflect the changes that have been made

   - documentation which is out of date is worse than none at all in some cases

CORRECTION

1) Use the system editor to correct the problem if you want to restart the program

   - allows you to continue by "→⎕*LC*" if desired

   - note that insertion of line labels or changing the header of the function will result in "*SI DAMAGE* " and an inability to restart

2) the system editor also allows a programmer to develop a line for insertion in a function, in immediate execution mode, and then insert it without retyping

   - this involves using ")nn" to edit the line and then, after entering the editor, ")" to retrieve it

   - avoiding unnecessary typing is important in keeping your train of thought and speeding up the debugging process

3) use 777 *QED* if you are not concerned about restarting the program and have major or complicated changes to make

   - this line-oriented general text editor works equally well on functions and documentation

   - again, *LIBRARY* 777 is a development library and programs may change without notice

   - documentation is on-line

4) use 777 *WSSEARCH* to make any extensive name replacements or modifications globally within a workspace

   - the major groups are *FN* and *SE*

   - remember to reassign any variable contentsto the new name before changing it globally

   - documentation is on-line

   eg. to change all references of '*FILEONE*' to '*DATASET1*' (where it holds a tie number)

```
SEFIND 'FILEONE'
DATASET1 ← FILEONE
SESHOW 'FILEONE'
'FILEONE' SERPLC BY 'DATASET1'
```

   eg. to show intermediate results of assignment to a variable '*LINES*' throughout the system

```
'LINES←' FNRPLC BY '⎕←'
```

5) use 777 *FNED* to edit functions which cannot be handled by normal means

   - a common example is having a large number of local variables named in the function header

   - this editor operates like the *CH* editor in ⍙ *EDIT* but edits functions

   - documentation is on-line

6) try alternate solutions

   - APL is renowned for the plethora of approaches to practically any problem

   - do not be afraid to try something new, just test it well

GOOD PROGRAMMING PRACTICES

SYSTEM DESIGN

1) strive for MODULARITY in design

   - APL is ideally suited to systems which are designed in this way

   - also saves time due to lack of duplication of effort

2) write TOP-DOWN systems

   - this allows good logical flow of control be the standard, rather than the exception

3) incorporate □*TRAP* (Event Trapping) where applicable to avoid unnecessary user interrupts

   - it is usually a good idea to plan your □*TRAP* 's but not implement them until needed

   - otherwise debugging can become difficult

4) try to keep functions short in length

   - absolute maximum should be 1 page, no more

   - difficult to read and understand when very long

5) do "WALK-THROUGH"'s of your logic and data flow before coding begins

   - this forces objectivity upon you the programmer

6) use UTILITY functions whenever possible since they have already been coded and tested

   - many good workspaces of utility functions can be found in *LIBRARY* 880

   eg. 880 *SAUCE*
   880 *TSSUTIL*

7) aim for consistency in your system design

   - try to handle similar problems the same way unless there is good reason to change

   - resulting system is then easier to use and debug

8) design in an *APL* fashion: minimize looping

   - usually more efficient and less prone to error

   - keeps functions clear, readable and concise

9) use only GLOBALS or ARGUMENTS to pass data between functions - NOT both

   - confusion results if both parameters and globals are used in the same function

   - harder to debug

10) choose your data representation and structure with care - it will govern the design of your entire system

   - decide on your data structure first, everything else will follow logically

CODING

1) be CONSISTENT in whatever style of coding you adopt

   - this will reduce the careless errors and mistakes that can be so costly

   - makes your system easier to read, understand and debug

2) use meaningfull names for variables, functions and files

   - common sense should be your guide here

   - good does not always mean long - just enough to convey its purpose

3) adopt a naming convention for objects in the workspace or system

   - allows a new programmer to catch on faster

   eg.

general utility	ΔWHATEVER
no concern of user	WHATEVER
file oriented	FWHATEVER
ordinary	WHATEVER

4) adopt a branching convention for functions

   - always test your condition at the top

   - this allows for the null case (initial pass)

   - use the same *APL* code for the branch

   eg.

normal	→(condition) ρ label
unless	→(condition) ↓ label
multiple	→(condition) / label, label, label

5) underscore line labels so they stand out in the function

   - again, consistency is crucial

   - try this and see how much more readable it is than before

6) always localize variables to the function in which they are used

   - shadowed values of variables are extremely difficult to catch when an error involves them

7) never change the value of your input data or function arguments

   - hampers restartability

   - poor style in programming

8) do not re-use variable or functions names for different purposes

   - confusion is again the obvious result

   - only exception might be *TEMP* or a loop counter

9) use only one exit point in a function

   - ties loose ends together

   - helps facilitate tracing the flow of control and calling of functions

   eg. *END*:→0

   - it may help to distinguish from a finish and an end when coding the function

10) write restartable code

   - no diamonds

   - no reassignment of a variable on a line where it is referenced

   - this topic is discussed in more detail in "RESTARTABILITY AND SECURE SYSTEMS" , another Advanced APL Topic

11) avoid temporary variables as much as possible

   - often the result can simply be passed to the left for further processing

TESTING

1) TEST for what you expect; then for everything else

   - this cannot be over-emphasized as a must for good programmers

2) be very careful with boundary areas on any problem

   - this is where the errors occur, at the edge conditions

3) ask other people to test the system once you are satisfied it is correct and ready to go into production

   - not good for the ego; excellent for the system - test

   - other people do not make your assumptions

4) test each module individually, even if it means constructing artificial dummy modules for sub-routines

   - one by one, each function can be verified correct and then basically disregarded from then on

5) if testing reveals bad code, do not patch it but rewrite it instead

   - patchwork functions grow worse with time and use

6) make sure the code "does nothing" gracefully

   - all too often this is the bane of a good program

   - it is especially important in *APL* with ⍳0 and ''

7) make it right before you make it faster
   make it fail-safe before you make it faster
   make it clear before you make it faster

   - once the code is clear, work on optimization (unless it is design flow)

DOCUMENTATION

1) KEEP IT SIMPLE STUPID (K I S S)

   - do not try to impress your reader, try to educate him/her

   - should consist of a) system documentation
     b) user documentation

2) USE EXAMPLES as much as possible

   - both pictures and examples are worth a thousand words

   - also much easier to do

3) comment your code briefly and succinctly but do not leave out any important details

   - no line of APL is unreadable if it is commented

   - remember the GOLDEN RULE ... would you want to inherit an uncommented system ?

4) make sure comments and code agree

   - when you modify the function, keep your comments up-to-date

5) do not comment bad code - rewrite it

   - it takes about the same time and is much more worthwhile

CONCLUSION

It takes time to become good at debugging, and often a lot of patience and determination, but it is worth it in the long run. The experience you gain during each encounter will save you time and money on the next one.

This seminar has tried to propose an approach to debugging which is general and yet efficient regardless of language or environment. In particular, we have concentrated on the *SHARP APL* programming environment and the advantages it has to offer the user. In addition, we have outlined some of the many tools presently available to aid in the debugging process.

All that remains is for you, the programmer, to remember to use these techniques. In the process, you will choose those that work for you, reject those that don't and develop your own techniques.

Best of Luck!